An integrated system for real-time Model Predictive Control of humanoid robots

Tom Erez, Kendall Lowrey, Yuval Tassa, Vikash Kumar, Svetoslav Kolev and Emanuel Todorov
University of Washington

***Abstract*— Generating diverse behaviors with a humanoid robot requires a mix of human supervision and automatic control. Ideally, the user's input is restricted to high-level instruction and guidance, and the controller is intelligent enough to accomplish the tasks autonomously. Here we describe an integrated system that achieves this goal. The automatic controller is based on real-time model-predictive control (MPC) applied to the full dynamics of the robot. This is possible due to the speed of our new physics engine (MuJoCo), the efficiency of our trajectory optimization algorithm, and the contact smoothing methods we have developed for the purpose of control optimization. In our system, the operator specifies subtasks by selecting from a menu of predefined cost functions, and optionally adjusting the mixing weights of the different cost terms in runtime. The resulting composite cost is sent to the MPC machinery which constructs a new locally-optimal time-varying linear feedback control law once every 30 msec, while planning 500 msec into the future. This control law is evaluated at 1 kHz to generate control signals for the robot, until the next control law becomes available. Performance is illustrated on a subset of the tasks from the DARPA Virtual Robotics Challenge.**

I. INTRODUCTION

Designing motor controllers for articulated robot platforms is difficult and time-consuming, often requiring control engineers to explicitly specify the motions for every task. The framework of Optimal Control seeks to automate the job of the control engineer through numerical optimization: the system designer specifies a simple high-level description of the required task (e.g., move forward, remain upright, bring an object) in terms of a *cost function*, and the low-level details of the movement that minimizes the cost emerge autonomously from the optimization process.

In *model-based* optimal control we provide a model of the robot's dynamics in addition to the cost function, and the optimization algorithm uses this model to predict the outcome of possible actions and find an optimal future plan. One realization of model-based optimal control is called Model-Predictive Control (MPC), an approach that relies on real-time trajectory optimization (section III). Applying optimization in an online fashion allows the robot to deal with deviations from the plan and generate robust behavior that reacts to changes in its environment. Focusing on the optimization of a single trajectory allows us to side-step the curse of dimensionality that constrains the search for globally-optimal policies via dynamic programming.

MPC is most commonly used in the chemical process control community [1]. In such domains the natural dynamics of the plant (e.g., distillation columns) is slow (in the order of minutes), and therefore online re-optimization is not a computational challenge. Similarly, online optimization is computationally straightforward when the system is low-dimensional; this enabled many previous applications of MPC in robotics, for example in the control of autonomous vehicles [2].

In contrast, humanoid robots are high-dimensional systems, and the timescale of the dynamics of such articulated robots is on the order of milliseconds. Therefore, online trajectory optimization for a humanoid is a significant computational challenge. This can be side-stepped by creating a reduced model, but the price is a loss of generality, since the model reduction process is manual and must be tailored to a specific domain. One successful example of such model reduction is the Spring-Loaded Inverted Pendulum (SLIP) model [3], which approximates the dynamics of a biped robot as a single point mass and the multi-joint leg as an inverted pendulum with a spring. However, while SLIP has been serving the legged robots community well, in general it is difficult and labor-intensive to craft model reductions for every task, and such reduced models can only be applied within a limited range of states and is unusable in a more general context (e.g., when the robot has to get up, leap, or push a heavy object).

In order to enable MPC to control a full-body humanoid without crafting special-purpose simplifications, the computational challenge must be tackled head-on. Our initial explorations of this domain [4] suggested that evaluating the dynamics, and in particular computing its derivatives, is the most significant computational bottleneck. Therefore, in the past few years we have focused on building a new physics engine that is specifically tailored for control and optimization of articulated robots, titled MuJoCo (**Mu**lti-**Jo**int dynamics with **Co**ntacts) [5].

In previous work, we used MuJoCo in several specific applications of simulated humanoid control (operating slower than real-time), including full-body stabilization [6] and hand manipulation [7]. However, the power of MPC lies in its versatility — its capacity to offer a common framework that can effectively control many different tasks. Here we present a framework for real-time control of a humanoid robot in a diverse set of tasks. The immediate motivation for this project was the DARPA Virtual Robotics Challenge (VRC), where teams competed in controlling a simulated humanoid robot. The robot and its environment are simulated using

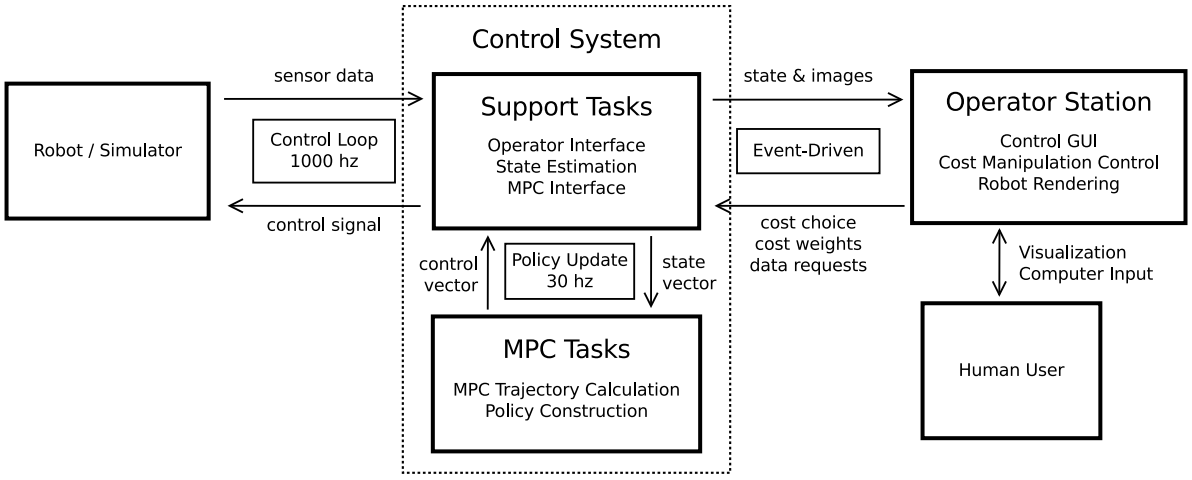


Fig. 1. An overview of the MPC control system. In the case of the VRC, the support tasks and MPC tasks ran on separate computers with a simulated robot, but this framework can be readily modified to perform on different system configurations. Timing characteristics, performance guarantees, and user experience are dependent on application and design of the robot, but data flow can remain unchanged for many use cases.

Gazebo/ODE[1] and controlled via Robot Operating System (ROS)[2] nodes. The challenge includes three different tasks: driving, diverse-terrain locomotion and manipulation. Since the VRC rules state that the simulated robot has no self-collisions (so the hand, for example, can penetrate the chest), we had to replicate that stipulation in our system. However, our system is perfectly capable of handling self-collisions.

The system we present here served as common platform to control the robot through all tasks, with a human operator in the loop providing high-level guidance to the system (section II) by manipulating the cost function being optimized, leaving all low-level control details to MPC.

To the best of our knowledge, this is the first paper to present a full integrated system for real-time application of MPC to a humanoid robot performing multiple tasks. Several contributions were made in order to achieve this goal: first, we employ our custom-build physics engine, MuJoCo [5], in the context of a computationally-efficient trajectory optimization framework that can run in real time (section III). Second, we developed a general-purpose machinery for specification of cost functions using residuals and norms (section IV). We developed a GUI system for operating the robot in real time, reflecting to the user the most updated state and receiving user inputs. Finally, in section V we describe the specifics of the cost functions we used to generate the behaviors of the VRC challenge, illustrated in the attached movie.

II. SYSTEM DESIGN

The system has three main components: the MPC optimization core (described in detail in section III), a suite of ROS nodes that interface to the robot system, and an operator interface. Integration of our MPC software with any robotic system requires a number of complementary processes, as illustrated in figure 1. For the VRC we based our suite in ROS, handling vision data, robot manipulator control, and most importantly, an interface to the MPC core which ran asynchronously on a dedicated computer.[3] Parallel to this control loop was a process that communicated with our software presented to a human operator outside of this network through event driven commands.

In this specific application, we were given a particular configuration with the simulated ATLAS robot ("the robot") streaming sensor data across a 10Gbps network to our software suite, which ran across two "field" computers — one running MPC, and the other doing estimation and communication. This second field computer uses callbacks in the ROS framework to handle all tasks not integral to the calculation of trajectories through MPC: caching of vision data and state estimation (section II-A). It is also through this framework that the operator requests data (such as a stereo pair of images) or makes parameter changes (such as commanding the MPC engine to switch between costs). Each ROS node had service routines that the operator interface process would call when commanded by the human user on a remote machine.

The separation of tasks between machines allows the MPC machinery to take full advantage of all resources to calculate trajectories. After state estimation is performed on the first machine, a state vector is sent to our MPC machinery, which consists of two main threads. First, a policy lookup thread that uses the state vector to interpolate a control signal from the current optimal trajectory. Second, the trajectory optimization thread that is described in more detail in section III. The first thread is meant to quickly provide a control signal, and in fact does so in under 200 microseconds, including state estimation and communication across the machines. After the control signal is returned to the first

[1]Available at http://gazebosim.org/.
[2]Available at http://ros.org/.

[3]Yet our MPC system is independent of ROS — it has no dependencies on any external libraries, and can run on both Linux, Windows and OSX.

machine, it is relayed back to the robot, thus closing our control loop.

The human operator can request visual information from the robot's cameras, dictate robot manipulator actions, and modify our MPC engine's behavior. While MPC offers powerful capabilities, it should always be possible to interject and guide the optimizer towards specific behaviors. The operator control computer presented a GUI to display the images, render the robot's state in a 3D window, and allow for cost function switching or weight changes. This combination of leveraging human knowledge and capability, along with MPC managing high frequency motor control, endows our humanoid robot with impressive capabilities.

A. State estimation

Our MPC machinery assumes that the current state of the dynamical system is either known exactly, or is being estimated by a separate process. The iLQG algorithm [6], which is the core of the MPC optimization (section III), treats the estimate as if it were the true state, and plans accordingly. Note that iLQG uses a linear-quadratic-Gaussian approximation to the optimal control problem, and is therefore blind to noise and uncertainty – in the sense that the control laws for a deterministic and a stochastic system with the same mean are identical. Thus estimation is really a separate process from MPC, and can be modified without affecting the rest of the system.

In the context of the VRC we designed a simple state estimator combining IMU data with a no-slip prior. The (simulated) IMUs provided drift-free orientation and angular velocity, and linear acceleration polluted with Gaussian noise with non-zero mean (i.e. bias). We first used a period without movement to calibrate the accelerometer bias. Then we integrated the accelerometer readings to obtain translational velocity and position. This resulted in some drift, which we corrected using a prior that the bodies contacting the ground are not slipping. This was done by applying forward kinematics and collision detection in the MuJoCo model, computing the contact-space velocities with the current estimate of the root velocity (and known joint velocities given by noise-free potentiometers), and correcting the estimated root velocity so that the contact-space velocities are reduced.

The resulting estimator was not perfect because Gazebo/ODE introduced unnatural spikes in the simulated accelerometer data, which were many standard deviations outside the specified accelerometer noise characteristics. At the same time, the simulated contacts did not fully stick even when they were supposed to (for the specified friction coefficient), thus our no-slip prior did not hold exactly. As a result, MPC was trying to correct imagined disturbances, and the corresponding corrections were themselves a disturbace – significantly degrading the overall performance of the system. These difficulties however are due to ODE simulation inaccuracies, and we do not expect them to occur when controlling a physical robot.

III. MODEL-PREDICTIVE CONTROL

Model Predictive Control (MPC), also known as online trajectory optimization or receding-horizon control, is a model-based control scheme. At every iteration, the current state of the robot is measured, and a trajectory optimization algorithm is applied to obtain a locally-optimal state-control trajectory emanating from the current state. The initial part of this trajectory is then used as a policy while the optimization is repeated. The trajectory optimizer is warm-started with the solution from the previous iteration, which greatly speeds up the method and often yields convergence after a single optimization step.

More formally, the discrete-time dynamics

$$\mathbf{x}_{i+1} = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i) \tag{1}$$

describe the evolution from time i to i+1 of the state $\mathbf{x} \in \mathbb{R}^n$, given the control $\mathbf{u} \in \mathbb{R}^m$. A trajectory is a sequence of states $\mathbf{X} \equiv \{\mathbf{x}_0, \mathbf{x}_1 \ldots, \mathbf{x}_N\}$ and controls $\mathbf{U} \equiv \{\mathbf{u}_0, \mathbf{u}_1 \ldots, \mathbf{u}_{N-1}\}$. The *total cost* J_0 is the sum of running costs ℓ and final cost ℓ_N, incurred when starting from $\mathbf{x}_0$ and applying $\mathbf{U}$ until the horizon N is reached:

$$J_0(\mathbf{x}_0, \mathbf{U}) = \sum_{i=0}^{N-1} \ell(\mathbf{x}_i, \mathbf{u}_i) + \ell_N(\mathbf{x}_N),$$

where the $\mathbf{x}_i$ for $i > 0$ are given by (1). The solution of the optimal control problem is the minimizing control sequence

$$\mathbf{U}^* \equiv \operatorname*{argmin}_{\mathbf{U}} J_0(\mathbf{x}_0, \mathbf{U}).$$

Note that in other contexts, trajectory optimization is often posed as the minimization

$$\min_{\mathbf{X},\mathbf{U}} J(\mathbf{X}, \mathbf{U}) \quad \text{s.t.} \quad \mathbf{x}_{i+1} = \mathbf{f}(\mathbf{x}_i, \mathbf{u}_i)$$

in other words, the entire state-control trajectory is subject to minimization. This type of trajectory optimization, called *direct optimization* is popular since it can be formulated as a generic sequential quadratic programming (SQP) problem and solved with off-the-shelf software. However in the MPC context, where the initial state constantly changes, it is not obvious how to warm-start the optimizer.

Letting $\mathbf{U}_i \equiv \{\mathbf{u}_i, \mathbf{u}_{i+1} \ldots, \mathbf{u}_{N-1}\}$ be the tail of the control sequence, the *cost-to-go* J_i is the partial sum of costs from i to N:

$$J_i(\mathbf{x}_i, \mathbf{U}_i) = \sum_{j=i}^{N-1} \ell(\mathbf{x}_j, \mathbf{u}_j) + \ell_N(\mathbf{x}_N).$$

The *Value* at time i is the optimal cost-to-go starting at $\mathbf{x}$:

$$V(\mathbf{x}, i) \equiv \min_{\mathbf{U}_i} J_i(\mathbf{x}, \mathbf{U}_i).$$

Setting $V(\mathbf{x}, N) \equiv \ell_N(\mathbf{x}_N)$, the Dynamic Programming Principle reduces the minimization over a sequence of controls $\mathbf{U}_i$, to a sequence of minimizations over a single control, proceeding backwards in time:

$$V(\mathbf{x}, i) = \min_{\mathbf{u}} [\ell(\mathbf{x}, \mathbf{u}) + V(\mathbf{f}(\mathbf{x}, \mathbf{u}), i+1)]. \tag{2}$$

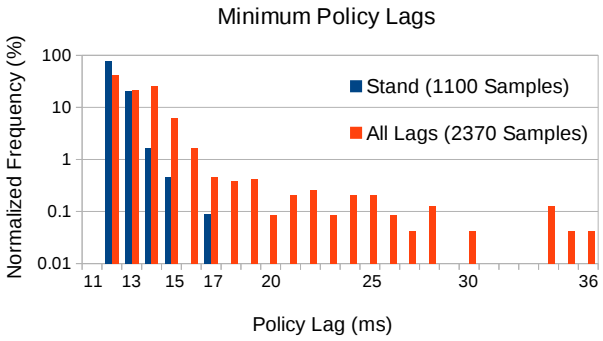


Fig. 2. Policy lag in MPC. This histogram demonstrates that policy lag can change depending on cost function, but still remains highly predictable and is a good measure of performance. No sample had a lag smaller than 11ms or bigger than 36ms. The "Stand" behavior induces a smaller and more reliable lag because the number of contacts never changes.

The trajectory optimizer which we used for this project is called iterative-LQG, which has been described in detail elsewhere [8] [6]. The algorithm proceeds by iterating a *forward pass* or *rollout* which integrates (1), followed by a *backward pass* which approximates a local solution to (2). Algorithm I provides a high-level overview of the iLQG module in our setup.

Algorithm I Trajectory optimization

Inputs: The dynamics $\mathbf{f}$, the running and final costs ℓ_i, ℓ_N, the current state $\mathbf{x}_0$ and the warm-start sequence $\mathbf{U}$.
Outputs: A locally-optimal control sequence $\mathbf{U}^*$.
1) Rollout: Integrate $\mathbf{U}$ to get the initial $(\mathbf{x}_i, \mathbf{u}_i)$ trajectory.
2) Derivatives: Compute the derivatives of ℓ and $\mathbf{f}$.
3) Backward Pass: Approximate a local 2nd-order solution to (2), obtain a U^* candidate.
4) Forward Pass: Integrate αU^* with several line search parameters $0 < \alpha < 1$ and pick the best one.

A. Timing

If a particular optimization iteration takes τ_i ms, a trajectory that emanates from the state at time t becomes available at time $t + \tau_i$. This trajectory is used for control in the next τ_{i+1} milliseconds, while a new trajectory (emanating from the state at time $t + \tau_i$) is being optimized. Therefore, the control signal along the first τ_i ms of the trajectory is never used to control the robot. Similarly, after $\tau_i + \tau_{i+1}$ ms this policy is replaced with a fresh one. Therefore, the controller only looks at the control values between time τ_i and τ_{i+1} along the trajectory. In general small value for the *policy lag* τ allows the optimizer to become aware sooner of unexpected changes to the state, and is critical for successful behavior of MPC.

The optimizer relies on the dynamical model to predict the future state of the system. In most cases this model is imperfect, and these modeling errors cause a mismatch between the optimizer's predictions and the system's real-world behavior. This mismatch grows as the integration time extends farther into the future. Yet, since every optimization iteration starts by querying the estimator for the robot's current state, the mismatch between prediction and reality at the initial part of the trajectory is expected to be small. Therefore, a small policy lag (smaller values of τ) is important for effective use of MPC.

In MuJoCo, the most intensive part of computing a single step is the handling of contacts. This was a source of timing variability — when more contacts are present (e.g., during manipulation or crawling tasks), the computation time (and therefore policy lag) grows. See figure 2 for a summary of our timing results.

The most time-consuming part of this algorithm is the finite-differencing of the dynamics at every integration time-step, but this part is simple to parallelize by sending every time step to a different processor core. Given that in this particular instance we had a 16-core machine available, all our trajectories had 16 time-steps, and the length of the planning horizon was adjusted by changing the length of the time step along the trajectory.

B. Design trade-offs

When choosing the parameters for the dynamics model, we had to balance several contradicting design goals: on the one hand, computing the dynamics should be fast, so as to provide low policy lags; on the other hand, the model should be accurate, so as to provide high-fidelity predictions of the system's dynamics; finally, the dynamics should be continuous and smooth, so as to provide useful gradients to the optimizer.

In order to achieve smooth dynamics we introduced a smoothing coefficient to the contact dynamics, which didn't predict accurately the robot's stiff collisions. This unrealistic dynamics was mitigated by the low policy lag that kept the plan synchronized with the robot's true state in the first part of the trajectory.

Another issue was the choice of the planning horizon. On the one hand, a small time-step provides more accurate predictions and a higher temporal resolution. On the other hand, a short planning horizon yields greedy, myopic behavior, and makes it difficult for the optimizer to discover more elaborate maneuvres (such as an autonomous getting-up sequence). In order to resolve this tension, we used a non-uniform time-step along the horizon: the first few time-steps were shorter (so as to provide high-resolution trajectory to the controller), while the rest were longer (since the low policy lag guaranteed that the latter part of the trajectory was never acted upon). See section V.

IV. COST FUNCTION DESIGN

The power of optimal control lies in the autonomous discovery of the detailed control policy given a high-level description of the task, formulated as a scalar cost function. However, not all cost functions are born equal. At one extreme is the sparse cost, where all states but a select few

incur a large penalty. In this case a long horizon and an exhaustive global search would be required to find the correct policy. On the other end of the spectrum, if we had access to the true value function, the optimal behavior can be found with a one-step greedy optimization.

When seeking the middle ground between these two extreme examples, we face a trade-off between the planning horizon and the level of detail of the cost function: in some cases we can computationally afford a long planning horizon (compared with the natural dynamics of the plant and the scope of the desired behavior) while still maintaining a small policy lag. In such a situation, the optimizer can discover effective behavior with simple, abstract cost functions, since the long-term effects of immediate actions are available. However, if the available computational power constrains us to a short planning horizon, we must design a more detailed cost function to help the optimizer discover good behavior. The danger of such an approach is overfitting – cost functions that are too specific for dealing with one scenario (e.g., walking on flat ground) may harm the robot's performance in a different scenario (e.g., uneven ground) since they overdetermine the details of the behavior and leave less room for creativity.

In summary, the structure of our cost functions must be flexible enough to allow us to specify both very general goals ("minimize the robot's angular momentum") and very specific ones ("bring the robot's hand to this location and orientation, while performing a grasp") with equal ease; letting us quickly find the sweet-spot in the aforementioned tradeoff. In addition to this design objective, an important technical restriction is that cost functions must be twice differentiable for the trajectory optimizer to succeed.

We chose a formulation with two entities: residuals and norms. A residual is a vector function of the state, and can be the result of kinematic computation (e.g., the Cartesian position of the hand) or dynamic ones (e.g., the reaction force between the foot and the ground). A *norm* is a scalar function of a residual vector. The cost function is a sum of terms, where every cost term is defined as the norm of a residual.

Formally, our cost structure is:

$$\ell(\mathbf{x}, \mathbf{u}) = \sum_{k=1}^{K} w_k f_k(\mathbf{r}_k(\mathbf{x}, \mathbf{u}))$$

Here the subscript k denotes one of K cost terms, each scaled by some weight $w_k \geq 0$. The functions $f_k()$ are the norms, simple twice differentiable scalar functions. The vector functions $\mathbf{r}_k()$ are the residuals. A desirable feature of this formulation is that it affords a computationally-cheap approximation for its derivatives, which are required for the iLQG optimization.

The Jacobians $\partial\mathbf{r}/\partial\mathbf{x}$ and $\partial\mathbf{r}/\partial\mathbf{u}$ can be obtained at a negligible computational cost. We can compute an analytic Jacobian for many quantities of interest, and for other residuals the Jacobian can be approximated by finite-differencing. Since we obtain the derivatives of the dynamics f by finite-differencing, we simply augment the dynamics with the residual $\mathbf{r}$ of our choosing, and approximate the Jacobians without any additional dynamics evaluations. The key benefit of this structure is that the residuals $\mathbf{r}$ can be arbitrarily complex without having to be analytically differentiable.

Given the Jacobians of the residuals and the derivatives of the norms we can obtain the exact gradient of every cost term and approximate its Hessians:

$$\frac{\partial \ell_k}{\partial \mathbf{x}} = w_k \frac{\partial f_k}{\partial \mathbf{r}} \frac{\partial \mathbf{r}}{\partial \mathbf{x}} \quad \text{and} \quad \frac{\partial^2 \ell_k}{\partial \mathbf{x}^2} \approx w_k \frac{\partial \mathbf{r}}{\partial \mathbf{x}}^{\mathsf{T}} \frac{\partial^2 f_k}{\partial \mathbf{r}^2} \frac{\partial \mathbf{r}}{\partial \mathbf{x}}$$

and similarly for derivatives w.r.t $\mathbf{u}$. The second expression involves the Gauss-Newton formulation, allowing us to approximate the cost derivatives without computing $\partial^2\mathbf{r}/\partial\mathbf{x}^2$, which would be computationally expensive to obtain in the general case.

As detailed in [6], our most useful norm function was the "smooth-abs" function $f(\mathbf{r}) = \sqrt{\mathbf{r}^{\mathsf{T}}\mathbf{r} + \alpha^2} - \alpha$. Because this function is linear outside of an α-sized neighborhood, the units of $\mathbf{r}$ (e.g. distance) are conserved, allowing for a more intuitive selection of the weights w. For torques and and other regularizing costs (see below), we tended to use the simple quadratic norm $f(\mathbf{r}) = \mathbf{r}^{\mathsf{T}}\mathbf{r}$.

Algorithm II Model-Predictive Control

Repeat indefinitely:

1) Estimate: Use the most recent sensor data to generate an estimate of the current state of the robot.

2) Transition: For every transition associated with the current cost, compute the associated norm and residual. If any such value goes below its threshold, transition to a new cost. If multiple thresholds are hit, choose the first transition (in the order they were defined).

3) [Alterations]: If a transition occurred, apply any associated alterations to the model.

4) User input: Check for user inputs, apply weight changes (if no transition occured) or switch to a new cost.

5) Trajectory optimization: [see algorithm I].

6) Update: Send the resulting control sequence $\mathbf{U}^*$ to the controller, which interpolates the control signal (according to time) at a high rate.

A. Cost transitions and alterations

In order to allow for more autonomy, we augmented the cost function system with a state machine that can switch between costs. For every cost, we may specify several conditions in terms of a threshold over a norm of some residual of the current state. If this condition is met, the system autonomously switches to some other cost. The general structure of the transitions may give rise to many interesting behaviors. In particular, we used it to design limit cycles and pre-defined behavioral sequences. Our walking behavior was made of a sequence of four behaviors: left leg swing, left-forward stance, right leg swing, and right-forward stance (see section V-B.3); our solution to the task of entering

the car involved a transition sequence (V-C.2). As opposed to other state-machine approaches to locomotion [9], [10], here we are switching between cost functions and not between explicit control laws. Note that such an abrupt change to the optimization target may well result in non-smooth control signal, but that did not disrupt the overall stability of the system in this case.

All the costs in the locomotion sequence also had a transition to standing, in case the robot came close enough to the target. Another sequence we designed is entering the car (section V-C.2). Here every transition depends on the success of the previous sub-task: grasping the car frame, setting a foot on the car's floor, etc.

We can also associate an alteration of the model with a transition event. Since there are many ways in which the parameters of the model affect the behavior, this feature can be used to serve different functions, as described in section V: for example, alterations were used to reposition the foot placement targets during walking (section V-B.3).

B. Class hierarchy of cost functions

In order to capture the diversity of tasks in the VRC in a succinct way, we organized the tasks and sub-tasks into a class hierarchy of cost functions.[4] At the most common level of the hierarchy we have terms that limit the space of likely behaviors (such as penalties for actuation and extreme acceleration of the head). At the next level we have the wide categories of the different task; for example, the different cost functions that were part of the walking and standing set of behaviors included the same core set of terms such as keeping upright, minimizing angular velocity of the pelvis, and so forth. Further down the hierarchy we had more specific behaviors: the manipulation-related cost functions inherited the standing stability terms and had additional terms for specifying specific desired positions for the end-effectors as specified in the next section.

V. RESULTS

Once the general MPC framework was built, our effort to develop robot behavior focused on the construction of the cost functions that encoded the various tasks of the challenge. Here we describe the various cost functions used by specifying the quantity that was minimized by every term in this function. As explained in section IV-B, the cost functions are organized in a class hierarchy of increasing specification, with many terms shared across multiple functions; therefore, this section is organized according to the different classes of behavior. Additional movies can be viewed at the project's website: homes.cs.washington.edu/~vikash/P_DRC.html

[4]Note the distinction between "class hierarchy", which implies inheritance of terms and coefficients, and "hierarchical optimization" (as in [11]), where multiple behavioral goals are specified with an order of precedence, and the optimizer must first satisfy the high-priority requirements before attending to the lower-level ones.

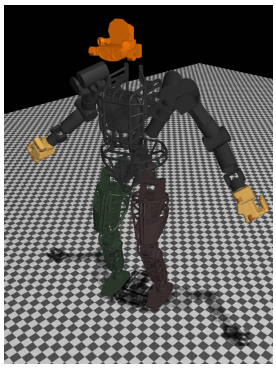


Fig. 3. Typical upright standing position.

A. Common cost terms

These terms form the base of the cost hierarchy and are common to all behaviors. The quantities minimized by these terms are:

- Joint torques
- Joint velocities
- Angular velocity of the pelvis
- Head acceleration (in cartesian coordinates)

All four terms use the quadratic norm, since our trajectory optimizer uses a local quadratic expansion of the cost and will therefore incur no approximation error w.r.t to these terms (section IV). Because torques are the control signals sent to the robot and are the output of the trajectory optimizer, the quadratic torque cost is the most important regularizing term. Independently, torques are also subject to per-actuator control-limits, given by the robot description.

B. Specific behaviors

In order to accomplish the different tasks of the VRC, we built several sets of cost functions that encoded specific behaviors: three locomotion modes (Walking, Slow Shuffling and Crawling), a sequence of costs that concludes in a robust standing pose, and a sequence of maneuvers that allow the robot to enter the car. Each behavior includes multiple cost functions that share certain terms, and all share the common terms mentioned above.

1) Standing up: Maintaining a stable stance (see figure 3) is critical for manipulation. For every term in this cost function we specify the quantity being minimized:

- STATIC STABILITY: this term penalizes the distance between the projection of the center of mass (CoM) and a line segment drawn between the feet (this line segment is an approximation to the support polygon).
- FACE FORWARD: penalizing the deviations between the orientation in XY plane of the pelvis, upper torso, and the two feet. This is computed by computing the difference between the relevant terms in the rotation matrix associated with the global orientation of each body.
- STAND UPRIGHT: deviations of the Z axis of upper torso and both feetfrom the global vertical direction.

- STAND HEIGHT: deviation of the global height of the upper torso from the fixed value of 1.2 m.

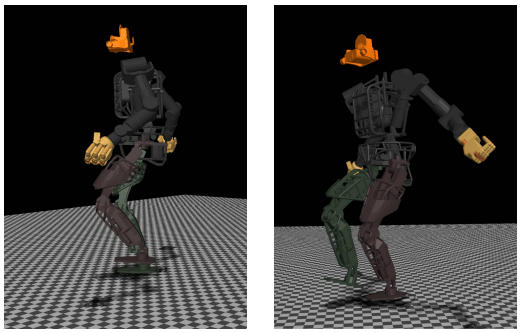
2) In-place shuffle: This behavior consists of two symmetric single-support states (L-stance and R-stance) that transition to each other (section IV-A) every 800 ms, causing the robot to step in place. Figure 4(a) shows the robot standing on its right leg. This behavior is used for multiple purposes: as transition between standing upright and walking behaviors, and as a stable and safe mode of locomotion in constrained narrow spaces. It inherits all terms of the standing upright behavior, but replaces the two-leg stability term with a similar single-leg term, penalizing the distance between the projection of the CoM and the foot. Additionally, it has the following terms:

- Single leg stance height: asking the robot to keep the swing leg's foot 10 cm over the other (to encourage the robot to stand on one foot).
- Orientation: penalizing deviations of the orientation of the pelvis and both feet from the direction of a user-specified target. This term prevents the swing leg from moving freely.

3) Walking: Walking is composed of a circular state machine of four states — right step, right stance, left step and left stance (see figure 4(b)). These costs inherit the terms of in-place shuffle, but override the STANCE HEIGHT term with a term that penalizes the distance of the swing foot from a foot target position. The position of this subject to model alterations upon cost transition from stance to step, repositioning the swing leg foot target at a pre-specified offset to the previous stance leg in the direction of the body orientation.

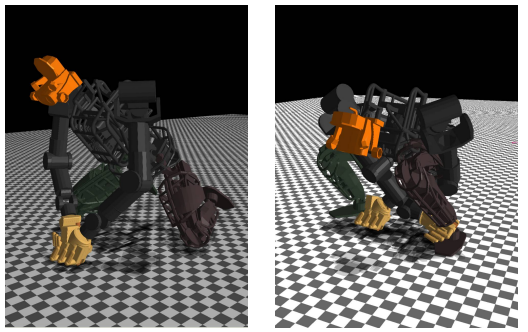
C. Pre-designed sequences

Some behaviors (such as entering the car) were too complicated to be discovered autonomously using the planning horizon available in the current implementation. In other cases (such as getting up), achieving the desired behavior with a short planning horizon led the robot to pursue unsafe behavior (e.g., springing up straight from laying on the ground) that had potential to fail and break the robot. In both cases, our solution is to manually decompose the task to a sequence of subtasks, and design a state-machine with transitions between several costs that guides the robot through a pre-designed behavior.



(a) Right stance (b) Right step

Fig. 4. Walking



(a) Kneeling, preparing to get into crouch pose. (b) Crouching, ready to get up.

Fig. 5. Getting up sequence

1) Getting up: Getting up from a fall consists of the following sequence:

I All fours: Stabilization with both hands and legs facing downwards.
II Kneel (figure 5(a)): This position is easy to get to from All Fours and is a natural transition to Crouch.
III Crouch (figure 5(b)): The last step before getting up. Note here we use the ZMP cost from Stand-Up.
IV Standing up as described in section V-B.3.

2) Entering the car: The sequence of entering the car included these sub-steps:

I Stand in front of the passenger door.
II Position the right hand on the car frame.
III Close the right hand and grab the car frame.
IV Send the left hand towards the steering wheel while raising the left leg onto the car's floor.
V Close the left hand and grab the steering wheel with the left hand.
VI Use the three contact points with the car (two grasping hands and left foot) to lift the body onto the seat.

The cost functions for items I-V were versions of the standing cost with additional terms for hand positions. Items IV and V involved switching the two-legged stability term with a single-leg stability term that allows the left leg to move freely. Item VI retains only the basic common terms, adding a term penalizing the distance of the pelvis from a target position on the seat.

D. Common subtask terms

This set of cost terms is shared among all cost functions, and allow the user more low-level control on the optimized behavior. The user selectively turns them on by temporarily increasing the corresponding weight. For every term, we specify the quantity that is being minimized:

- FACE LOOK-AT: offset between the face-forward vector and the unit vector pointing from the pelvis to the look-at target (offsets between vectors are computed as the sum of differences of the respective rotation matrices).

- HAND LOOK-AT: offset between the left or right (L/R) hand camera vector and the unit vector pointing from L/R palm to the look-at target.
- REACH HEAD-TARGET: distance between the head and a designated head-target in XY plane. Such targets are interactively positioned by the user.
- HAND REACH: translation and orientation offset between the L/R hand and its respective hand-target.

All these terms are initialized with a zero weight, and the weights are not shared across the cost functions. This allows us to use these terms only when needed, and are meant to complement existing behavior (e.g., walking or standing). Choosing the weight is an empirical process of gradually increasing the weight through manual tuning while observing the resulting behavior. Immediate feedback enables very fine control over the behaviors during the tuning process. Usually mild weights are enough to induce the appropriate behaviour. Here are some examples of use cases for these terms beyond their immediate purpose:

- STANDING UPRIGHT + FACE LOOK-AT: when used with an initially-wide look-at offset angle, this term induces in-place turning of the entire body.
- IN-PLACE SHUFFLE + FACE LOOK-AT: Enables gradual and smooth in-place turning. This was very useful for fine corrections in body orientation.
- IN-PLACE SHUFFLE + HEAD REACH: Induces slow walking (~5mm/sec) in the forward direction of the body. Useful for fine grained and careful re-localization in narrow and constrained environments.
- IN-PLACE SHUFFLE + HAND REACH: Induces slow walking (~5mm/sec) in the direction of hand target while maintaining body orientation. Useful for fine grained and careful re-localization in narrow and constrained environments when object of interest is off-reach.
- STAND + HAND REACH: Forces a step in the target direction when the object of interest is out of reach.

VI. CONCLUSION

This paper describes an integrated system for controlling humanoid robots via full-body MPC, augmented with high-level human guidance in the form of cost function specification. While the system was developed in the context of the DARPA Virtual Robotics Challenge, it is quite universal and we will soon apply it to physical robots in other contexts. We have previously used MPC to generate rich robotic movements, however this was done slower than real-time and the simulated robots were not as complex as the Atlas humanoid. Our attempts to develop a real-time MPC system for a high-degree-of-freedom robot revealed that, with existing computing speed, the planning horizon cannot be made long enough to discover complex movements with the simple and abstract costs we prefer to use. Thus we had to develop an elaborate machinery for cost function design and online cost switching, which allowed us to specify subtasks and sequence them as needed. Nevertheless, designing these more elaborate costs is still much faster than the labor-intensive work that goes into designing control laws directly. Once our machinery was ready, the cost functions generating all the behaviors illustrated in this paper were designed and fine-tuned by a team of three people in about three weeks.

As in our previous work, we obseved that MPC is very robust to model errors – in this case caused by discrepancies between the MuJoCo model used for planning, and the Gazebo/ODE model used for simulation. Performance degraded significantly in the presence of large state estimation errors (caused by simulation inaccuracies in ODE), but this issue is specific to the VRC context and is unlikely to arise when working with physical robots equipped with modern sensors, or with more accurate simulations.

ACKNOWLEDGEMENTS

This research was funded by DARPA and NSF.

REFERENCES

[1] F. Allgower, R. Findeisen, and Z. K. Nagy, "Nonlinear model predictive control: from theory to application," *Chinese Institute of Chemical Engineers*, vol. 35, no. 3, pp. 299–316, 2004.

[2] R. González, M. Fiacchini, J. L. Guzmán, T. Álamo, and F. Rodríguez, "Robust tube-based predictive control for mobile robots in off-road conditions," *Robotics and Autonomous Systems*, vol. 59, no. 10, pp. 711–726, 2011.

[3] R. Blickhan, "The spring-mass model for running and hopping," *Journal of Biomechanics*, vol. 22, pp. 1217–1227, 1989.

[4] Y. Tassa, T. Erez, and W. Smart, "Receding horizon differential dynamic programming," in *Advances in Neural Information Processing Systems 20*, J. Platt, D. Koller, Y. Singer, and S. Roweis, Eds. Cambridge, MA: MIT Press, 2008, p. 1465.

[5] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: a physics engine for model-based control," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS)*, 2012.

[6] Y. Tassa, T. Erez, and E. Todorov, "Synthesis and stabilization of complex behaviors through online trajectory optimization," in *IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS)*, 2012.

[7] ——, "Control-limited differential dynamic programming," *Under Review*.

[8] E. Todorov and W. Li, "A generalized iterative LQG method for locally-optimal feedback control of constrained nonlinear stochastic systems," in *Proceedings of the 2005, American Control Conference, 2005.*, Portland, OR, USA, 2005, pp. 300–306.

[9] M. Raibert, *Legged Robots that Balance*. MIT Press, 1986.

[10] U. Muico, J. Popović, and Z. Popović, "Composite control of physically simulated characters," *ACM Transactions on Graphics*, vol. 30, no. 3, 2011.

[11] A. Escande, N. Mansard, and P.-B. Wieber, "Hierarchical quadratic programming," *International Journal of Robotics Research*, October 2012, [submitted].